A Multi-level Optimization Strategy to Improve the Performance of Stencil Computation

Gauthier Sornet, Fabrice Dupros, and Sylvain Jubertie

Univ. Orléans, INSA Centre Val de Loire, LIFO and BRGM.
Motivations

- Many applications widely use stencil computation:
  - Image: Convolution filters
  - PDE solvers: Fullswof, EFISPEC 3D,...
- Stencil computation represents a major part of the total computing elapsed time.
- Stencil computation has a low arithmetical intensity.
- Compilers optimisations could be limited: unrolling, vectorization...
Outline

1. 7pts and 27pts Stencils
2. Reuse intensity
3. Optimizations
4. The Pochoir library
5. Experimental results
6. Conclusion and perspectives
Gauthier Sornet, Fabrice Dupros, and Sylvain Jubertie
A Multi-level Optimization Strategy to Improve the Performance of the Stencil Computation

7pts and 27pts Stencils
Reuse intensity

Characterization

Stencil has a low compute / memory ratio. We need a metric to characterize stencil types.

Definition

The reuse intensity is the factor between memory reuse times and the number of loaded bytes for each iteration.

Exemple
## Reuse intensity

### R.I of the stencil under study

<table>
<thead>
<tr>
<th>Representation</th>
<th>Name</th>
<th>Number of cells</th>
<th>Number of operations by cell</th>
<th>R.I</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="7pts" /></td>
<td>7pts</td>
<td>7</td>
<td>1</td>
<td>$7 \times \frac{1}{4} = 1.75$</td>
</tr>
<tr>
<td><img src="image" alt="27pts" /></td>
<td>27pts</td>
<td>27</td>
<td>1</td>
<td>$27 \times \frac{1}{4} = 6.75$</td>
</tr>
</tbody>
</table>
Manual Vectorization

Compilers do not guarantee vectorization. We explicitly use SIMD instructions to force the compiler to use it and to optimize with it in our way.

Our computing platforms are able to compute 8 floats of 32 bits by AVX instruction.

SISD add (not vectorized instructions):

5 cells added at a time.

SIMD add (vectorized instructions):

8*5 cells added at a time.

Theoretical speed up of 8.
Optimizerizations

- Crossing domain with tiling
- Space tiling
- Time tiling
- Memory, Cache optimisations

- Cells in caches
- Cells lost from caches
- Cells loaded twice

Gauthier Sornet, Fabrice Dupros, and Sylvain Jubertie
A Multi-level Optimization Strategy to Improve the Performance of the Stencil Computation
## Optimizations

### Stencil Composition

<table>
<thead>
<tr>
<th>Representation</th>
<th>Name</th>
<th>Number of cells</th>
<th>Number of operations by cell</th>
<th>R.I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7pts</td>
<td>7</td>
<td>1</td>
<td>7*1/4 = 1.75</td>
</tr>
<tr>
<td></td>
<td>25pts</td>
<td>25</td>
<td>1</td>
<td>25*1/4 = 6.25</td>
</tr>
</tbody>
</table>

R.I of the stencil under study
Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson.
The pochoir stencil compiler. SPAA ’11.
The Pochoir library

Two optimization levels:

➢ Independant Cilk tasks cutting to reduce thread synchronisation
➢ Time and space cutting to optimise caches

Let consider a one x dimension domain to compute $t$ times:

➢ What about vectorization?
Experimental results

Architectures

➢ Intel Xeon E5-2697 v2 Ivybridge processors, for a total of twenty-four cores at 2.7Ghz.
➢ Intel Xeon E5-2697 v4 Broadwell processors, for a total of eighteen cores at 2.3Ghz.

Compilers

Clang 3.8, GCC 6.2 and ICC 17 with -O3 -march=native optimization flags and OpenMP multi-threading.

Fixed parameters

➢ Single float precision
➢ 512x512x512 grid
➢ 100 iterations
➢ Each results from 10 runs
Experimental results

Rooflines

Experimental Ivybridge roofline:
- 427.53 GFLOPS
- Dram = 59.24 GB/s

Experimental Broadwell roofline:
- 556.52 GFLOPS
- Dram = 66.23 GB/s
Experimental results

Performances
Experimental results

Rooflines

Experimental Ivybridge roofline

- Dram - 59.24 GB/s
- 427.53 GFLOPS
- 130.14 GFLOPS
- 127.66 GFLOPS
- 34.56 GFLOPS
- 1.75 GFLOPS
- 6.25 GFLOPS
- 6.75 GFLOPS
- 7.22 GFLOPS

Experimental Broadwell roofline

- Dram - 66.23 GB/s
- 556.52 GFLOPS
- 141.78 GFLOPS
- 41.02 GFLOPS
- 1.75 GFLOPS
- 6.25 GFLOPS
- 6.75 GFLOPS
- 8.4 GFLOPS
#### Experimental results

**Elapsed time**

<table>
<thead>
<tr>
<th></th>
<th>Clang3.8</th>
<th>ICC17</th>
<th>GCC6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>7pts 100it</strong></td>
<td>1875ms</td>
<td>1956ms</td>
<td>2117ms</td>
</tr>
<tr>
<td><strong>25pts 50it</strong></td>
<td>937,5ms</td>
<td>1160ms</td>
<td>1276ms</td>
</tr>
<tr>
<td><strong>Poch 7pts 100it</strong></td>
<td></td>
<td>2809,55ms</td>
<td></td>
</tr>
<tr>
<td><strong>Poch 25pts 50it</strong></td>
<td></td>
<td>2300,745ms</td>
<td></td>
</tr>
</tbody>
</table>

**Broadwell**

<table>
<thead>
<tr>
<th></th>
<th>Clang3.8</th>
<th>ICC17</th>
<th>GCC6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>7pts 100it</strong></td>
<td>2157ms</td>
<td>2288ms</td>
<td>2174ms</td>
</tr>
<tr>
<td><strong>25pts 50it</strong></td>
<td>1519,5ms</td>
<td>1305,5ms</td>
<td>1338,5ms</td>
</tr>
<tr>
<td><strong>Poch 7pts 100it</strong></td>
<td></td>
<td>1315,62ms</td>
<td></td>
</tr>
<tr>
<td><strong>Poch 25pts 50it</strong></td>
<td></td>
<td>2534,22ms</td>
<td></td>
</tr>
</tbody>
</table>

**Ivybridge**

<table>
<thead>
<tr>
<th></th>
<th>Clang3.8</th>
<th>ICC17</th>
<th>GCC6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>7pts 100it</strong></td>
<td>2157ms</td>
<td>2288ms</td>
<td>2174ms</td>
</tr>
<tr>
<td><strong>25pts 50it</strong></td>
<td>1519,5ms</td>
<td>1305,5ms</td>
<td>1338,5ms</td>
</tr>
<tr>
<td><strong>Poch 7pts 100it</strong></td>
<td></td>
<td>1315,62ms</td>
<td></td>
</tr>
<tr>
<td><strong>Poch 25pts 50it</strong></td>
<td></td>
<td>2534,22ms</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

➢ Simple optimizations but hard to make them work together.
➢ Experimental results follow the R.I analyses.
➢ Pochoir relies on ICC vectorization and optimization.
➢ The stencil composition is interesting.

Perspectives

➢ Automatic optimization.
➢ DSL or DSEL.
➢ Specific compiler or meta-programming.
Any question?